What Is Claimed Is:

1. A code generation method, comprising:

generating a table of patterns, each pattern in the table comprising an FMA (fused multiply-add) DAG (Directed Acyclic Graph), a canonical form equivalent of the FMA DAG, and a shape corresponding to the canonical form equivalent; and

matching incoming floating point expressions against the patterns in the table of patterns during compilation of a program.

2. The method of claim 1, wherein generating the table of patterns occurs once during compilation of a compiler.

3. The method of claim 1, wherein the FMA DAG comprises a sequence of FMA instructions that form a Directed Acyclic Graph.

4. The method of claim 3, wherein arguments for each instruction in the sequence of FMA instructions comprise terminals a, b, c, ... and constants one (1) and zero (0), wherein each terminal appears once in the sequence of FMA instructions and the FMA DAG includes at least one node.

5. The method of claim 1, wherein the canonical form equivalent of the FMA DAG comprises a sum of products of the terminals, wherein all of the terminals are sorted within a product and all products are sorted lexicographically.

6. The method of claim 1, wherein the shape comprises a binary representation of the canonical form equivalent in which all terminals in the canonical form equivalent are replaced with a binary "1" and all operation signs in the canonical form equivalent are replaced with a binary "0".

7. The method of claim 1, wherein generating a table of patterns comprises:

generating all possible FMA DAGs of a predefined complexity or less;

determining canonical forms and shapes for each FMA DAG;

sorting the generated FMA DAGs according to shape;

pruning the generated FMA DAGs;

sorting each group of shapes according to complexity and height;

encoding each pattern into a 64-bit number; and

storing the patterns as a table in a file.

8. The method of claim 7, wherein generating all possible FMA DAGs of a predetermined complexity or less includes generating all possible FMA DAGs of complexity 5 or less.

9. The method of claim 7, wherein shapes are handled as integers written in binary form.

10. The method of claim 7, wherein pruning the generated FMA DAGs comprises eliminating duplicate FMA DAGs and sub-optimal FMA DAGs.

11. The method of claim 10, wherein duplicate FMA DAGs comprise DAGs which have the same canonical form, the same complexity, and the same height.

12. The method of claim 11, wherein complexity comprises the number of FMA instructions in the FMA DAG and height comprises the number of levels in the FMA DAG.

13. The method of claim 1, wherein matching incoming floating point expressions against the patterns in the table of patterns during compilation of a program comprises:

determining a canonical form and shape for an incoming floating-point expression;

finding a pattern in the table of generated patterns that has the same shape as the incoming floating-point expression and at least as many terminals as the incoming floating-point expression;

determining whether a valid mapping exists between formal terminals and actual terminals, wherein formal terminals are terminals from the pattern that was found and actual terminals are terminals from the canonical form of the incoming floating-point expression; and

if the mapping is valid, then replacing the terminals in the corresponding FMA DAG with the actual terminals and determining sign combinations to find the correct sign combination and canonical form of the DAG equal to the incoming expression.

14. The method of claim 13, further comprising if it is determined that a valid mapping does not exist, then repeating the finding process and the valid mapping determination process until the mapping is valid.

15. The method of claim 13, wherein if the mapping is valid, the method further comprising providing an optimal sequence of FMA (fused multiply-add), FMS (fused multiply-subtract), and/or FNMA (fused negate multiply-add) instructions as compiled code for computing the incoming expression.

16. The method of claim 15, wherein the optimal sequence of FMA, FMS, and/or FNMA instructions comprise minimal complexity, minimal latency, and argument availability, wherein minimal complexity requires the number of instructions in the sequence of instructions to be minimal, wherein minimal latency requires the height of the DAG to be minimal when compared to all possible DAGs with minimal complexity, and wherein argument availability requires smaller terminals to be placed as close to the root node of the DAG as possible while still preserving the minimal complexity and the minimal latency when a strict order is placed on the set of terminals in the DAG.

17. An article comprising: a storage medium having a plurality of machine accessible instructions, wherein when the instructions are executed by a processor, the instructions provide for generating a table of patterns, each pattern in the table comprising an FMA (fused multiply-add) DAG (Directed Acyclic Graph), a canonical form equivalent of the FMA DAG, and a shape corresponding to the canonical form equivalent; and

matching incoming floating point expressions against the patterns in the table of patterns during compilation of a program.

18. The article of claim 17, wherein generating the table of patterns occurs once during compilation of a compiler.

19. The article of claim 17, wherein the FMA DAG comprises a sequence of FMA instructions that form a Directed Acyclic Graph.

20. The article of claim 19, wherein arguments for each instruction in the sequence of FMA instructions comprise terminals a, b, c, ... and constants one (1) and zero (0), wherein each terminal appears once in the sequence of FMA instructions and the FMA DAG includes at least one node.

21. The article of claim 17, wherein the canonical form equivalent of the FMA DAG comprises a sum of products of the terminals.

22. The article of claim 17, wherein the shape comprises a binary representation of the canonical form equivalent in which all terminals in the canonical form equivalent are replaced with a binary "1" and all operation signs in the canonical form equivalent are replaced with a binary "0".

23. The article of claim 17, wherein instructions for generating a table of patterns comprises instructions for:

generating all possible FMA DAGs of a predefined complexity or less;

determining canonical forms and shapes for each FMA DAG;

sorting the generated FMA DAGs according to shape;

pruning the generated FMA DAGs;

sorting each group of shapes according to complexity and height;

encoding each pattern into a 64-bit number; and

storing the patterns as a table in a file.

24. The article of claim 23, wherein instructions for generating all possible FMA DAGs of a predetermined complexity or less includes instructions for generating all possible FMA DAGs of complexity 5 or less.

25. The article of claim 23, wherein shapes are handled as integers written in binary form.

26. The article of claim 23, wherein instructions for pruning the generated FMA DAGs comprises instructions for eliminating duplicate FMA DAGs and sub-optimal FMA DAGs.

27. The article of claim 26, wherein duplicate FMA DAGs comprise DAGs which have the same canonical form, the same complexity, and the same height.

28. The article of claim 27, wherein complexity comprises the number of FMA instructions in the FMA DAG and height comprises the number of levels in the FMA DAG.

29. The article of claim 17, wherein instructions for matching incoming floating point expressions against the patterns in the table of patterns during compilation of a program comprises instructions for:

determining a canonical form and shape for an incoming floating-point expression;

finding a pattern in the table of generated patterns that has the same shape as the incoming floating-point expression and at least as many terminals as the incoming floating-point expression;

determining whether a valid mapping exists between formal terminals and actual terminals, wherein formal terminals are terminals from the pattern that was found and actual terminals are terminals from the canonical form of the incoming floating-point expression; and

if the mapping is valid, then replacing the terminals in the corresponding FMA DAG with the actual terminals and determining sign combinations to find the correct sign combination and canonical form of the DAG equal to the incoming expression.

30. The article of claim 29, further comprising instructions for if it is determined that a valid mapping does not exist, then repeating the finding process and the valid mapping determination process until the mapping is valid.

31. The article of claim 29, wherein if the mapping is valid, the method further comprising instructions for providing an optimal sequence of FMA (fused multiply-add), FMS (fused multiply-subtract), and/or FNMA (fused negate multiply-add) instructions as compiled code for computing the incoming expression.

32. The article of claim 31, wherein the optimal sequence of FMA, FMS, and/or FNMA instructions comprise minimal complexity, minimal latency, and argument availability, wherein minimal complexity requires the number of instructions in the sequence of instructions to be minimal, wherein minimal latency requires the height of the DAG to be minimal when compared to all possible DAGs with minimal complexity, and wherein argument availability requires smaller terminals to be placed as close to the root node of the DAG as possible while still preserving the minimal complexity and the minimal latency when a strict order is placed on the set of terminals in the DAG.

33. A code generation system, comprising:

a processor having an instructions set comprising fused instructions;

a memory, the memory comprising a code generator having a floating-point module coupled to an optimizer and a table of patterns coupled to the optimizer, the processor for enabling the code generator to receive floating-point expressions and to generate a sequence of optimal fused multiply-add, fused multiply-subtract, and/or fused negate multiply-add instructions to compute the floating-point instruction.

34. The system of claim 33, wherein the processor to enable the floating-point module to receive as input source code and to extract floating-point expressions from the source code.

35. The system of claim 33, wherein the processor to enable the optimizer to receive the floating-point expression from the floating-point module and to determine a canonical form and shape for the input floating-point expression.

36. The system of claim 35, wherein the processor to further enable the optimizer to search the table of patterns to find a pattern having a canonical form, shape, and at least an equivalent amount of terminals to that of the canonical form, shape, and terminals of the input floating-point expression.

37. The system of claim 36, wherein the processor to further enable the optimizer to determine whether a valid mapping exists between the terminals of the

pattern and the terminals of the input floating-point expression, and if there is a valid mapping, the processor to further enable the optimizer to replace the terminals in the corresponding FMA DAG with the terminals from the input floating-point expression and to determine sign combinations to find a correct sign combination and canonical form of the DAG equal to the incoming expression.

38. The system of claim 37, wherein the processor to further enable the optimizer to provide an optimal sequence of FMA (fused multiply-add), FMS (fused multiply-subtract), and/or FNMA (fused negate multiply-add) instructions based on the correct sign combination and canonical form of the DAG as compiled code for computing the incoming expression.

39. The system of claim 38, wherein the optimal sequence of FMA, FMS, and/or FNMA instructions comprise minimal complexity, minimal latency, and argument availability, wherein minimal complexity requires the number of instructions in the sequence of instructions to be minimal, wherein minimal latency requires the height of the DAG to be minimal when compared to all possible DAGs with minimal complexity, and wherein argument availability requires smaller terminals to be placed as close to the root node of the DAG as possible while still preserving the minimal complexity and the minimal latency when a strict order is placed on the set of terminals in the DAG.